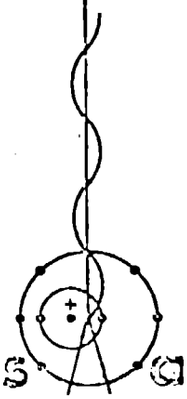
LA-UR -78-1849

conf-790113--5

TITLE: AUTOMATIC PROGRAM GENERATION: FUTURE OF SOFTWARE ENGINEERING

AUTHOR(S): JOEL H. ROBINSON, E-5

SUBMITTED TO: Dr. Richard E. Fairley
Computer Science Department
Colorado State University
Fort Collins, Colorado 80521



los alamos
scientific laboratory
of the University of California
LOS ALAMOS, NEW MEXICO 87545

An Affirmative Action/Equal Opportunity Employer

Form No. 836 R2
St. No. 2629
1/78

DEPARTMENT OF ENERGY
CONTRACT W-7405-ENG. 36

MASTER

AUTOMATIC PROGRAM GENERATION: FUTURE OF SOFTWARE ENGINEERING

by

Joel H. Robinson
Group E-5, MS-582
Los Alamos Scientific Laboratory
University of California
Los Alamos, New Mexico 87545
Phone: (505)-667-7100 or
(505)-667-3125

Abstract

At this moment software development is still more of an art than an engineering discipline. Each piece of software is lovingly engineered, nurtured, and presented to the world as a tribute to the writer's skill. When will this change? When will the craftsmanship be removed and the programs be turned out like so many automobiles from an assembly line? Sooner or later it will happen, economic necessities will demand it.

With the advent of cheap microcomputers and ever more powerful super computers doubling capacity we must produce much more software. Our choices are to double the number of programmers, double the efficiency of each programmer, or find a way to automatically produce the needed software. Producing software automatically is the only logical choice.

How will automatic programming come about? Some of the preliminary actions which need to be done and are being done are: encourage programmer plagiarism of existing software through public library mechanisms; produce well understood packages such as compilers automatically; develop languages capable of producing software as output; and learn enough about the whole process of programming to be able to automate it. Clearly, the emphasis must not be on efficiency or size since ever larger and faster hardware is coming.

I. INTRODUCTION

In the future, we will speak to computers and they will perform the tasks commanded and answer us. They will be able to tell us of problems they detect and we will tell them what action to take. Super computers will be able to handle a wide variety of questions, providing us with massive data resources. Microcomputers will be everywhere doing the same things, only slower and on a lesser scale. When they do not understand a command, they will ask for further clarification and instruction.

The science fiction in the above paragraph is not in the audio interface; that has been done. Nor is the fiction in the hardware interfacing to many different peripheral devices; that has been done. The fiction is not in computer speed or size; both will continue to increase while decreasing in cost. The science fiction is in the software. No super computer runs without programs, and no means of automatically generating computer programs of a diverse nature exists today. It is likely that none is about to appear.

Computer hardware capability is increasing faster than computer software capability. The gigantic strides in hardware development that have come and are certain to continue to come are not being met by giant strides in software. Exactly how bad this problem is and some of the solutions are explored in this paper.

II. HOW BAD IS THE PROBLEM?

Software used to only be about 10% of the cost of computing. It is now roughly more than equal the cost of hardware and projections for the middle 1980s could put software at 900%. Past that smart, manufacturers could give away the super computers and charge for the rights to write software! Microcomputers are already in the situation of being far less valuable than the software which runs on them. When will this change? How will programs be mass produced at low cost?

New generations of super computers seem to be appearing at less than a decade apart. Great improvements within generations appear every few years. Micro and minicomputers are evolving faster than that. Conversely, the software looks similar from generation to generation. Reliable software for a super computer takes a long time to develop, it is expensive, somewhat debugged, and usually late. Micro and minicomputers share the same problem to a lesser extent. When will this problem disappear? How can software for new computers be available at the same time as the hardware?

These scourges of computing must be faced squarely and defeated. For computing power to advance to the science fiction dreams of today, the methods of producing software must change.

III. THE ART OF PROGRAMMING

Unfortunately, many people, particularly programmers, still feel that programming should be an artistic expression. While it can be performed that way, efficiency of production suffers. Trying to squeak the last machine cycle out a computer, whether super computer or microcomputer, is not usually cost justified today. Hardware efficiency can lead you to the position of a team of efficiency experts reviewing a concert and reporting:

> "For considerable periods the four oboe players had nothing to do. Their numbers should be reduced, and the work spread more evenly over the whole of the concert, thus eliminating peaks of activity. ... All the twelve first violins were playing identical notes. This seems unnecessary multiplication. The staff of this section should be drastically cut; if a large volume of sound is required, it could be obtained by means of electronic amplifiers. ... There seems to be too much repetition of some musical passages. Scores should be drastically pruned. No useful purpose is served by repeating on the horns a passage which has already been handled by the strings. It is estimated that if all redundant passages were eliminated the whole concert time of two hours could be reduced to twenty minutes, and there would be no need for an interval."[1]

1 Branscomb, pp XVIII-XIX

IV. CURRENT SOLUTIONS

Won't today's methods of programming suffice to evolve into automatic programming? The typical approach to solving a programming crisis is to put more programmers and/or better programmers on the task. While these approaches do have a fair success history, they are limited by human effort. Obviously not everyone has the interest or aptitude to program using today's methods and languages. There is a large segment of the world population that is unwilling to sit at a desk or CRT for 8 hours a day talking to a computer. It is also obvious that programmer productivity can not keep pace with hardware improvements. In fact, not much has changed at all since the development of the last generation's "automatic programming tool," compilers! With all the laudable increases in programmer productivity from the new methodologies, the increases are not enough. There are current reports of programmer productivity for debugged code of 40 lines per day (Linger and Mills) with other averages being higher or lower. Human ability will soon set a ceiling on programmer productivity using today's nonautomated tools. After that the process must be automated in part or in pieces to achieve the needed software. While it is possible to double the number of programmers in the U.S., that doubling would only take care of the next doubling of hardware capability and software demands.

The developed countries of the world cannot afford to continually remove so many people from other occupations. Economic demands will soon force automation in programming.

V. STEPS TO AUTOMATION

The road to complete automatic program generation may never be taken. There is, however, enough demand for at least progress to lay out how the course should be started.

One of the first steps that is going to have to occur is standardization of software languages. Without it, progress made in one dialect or one obscure language is of very little help to anyone else. There are hundreds of distinguishable dialects and languages impeding programming progress. As was quoted earlier, "No useful purpose is served by repeating on the horns a passage which has already been handled by the strings."[2] Note the U. S. Government has a hard time making sure that all the different COBOL's around work the same. The painful but profitable steps of eliminating dialects and consolidating languages is of paramount importance to eve thinking about automatic program generation. There is a definite historical precedent that we should heed:

> "Then the Lord came down to see the city and tower which mortal men had built, and he said, 'Here they are, one people with a single language, and now they have started to do this; henceforward nothing they have a mind to do will be beyond their reach. Come let us go down there and confuse their speech, so they will not understand what they say to one another:' So the Lord dispersed them from there all over the earth and they left off building the city." GEN. 11:5-9

If we want to accomplish progress, we must not hinder ourselves with many languages.

---

2 Branscomb, IBID.

Once computer professionals are speaking the same languages regardless of machine type, then they can begin to share software tools. The idea of patenting software so that no one else may advance using your work is detrimental to the industry. Certainly copyrights will not interfere with progress if used to protect proprietary information, because one can still use a written idea without infringing the copyright. Software protection needs to be worked out carefully in companies that make a living selling software, but it can be done. All technological progress since the invention of the wheel has been based on not reinventing it each time you want to use one!

Once the computer industry has agreed to the premise that languages should be standardized and/or limited, the question arises as to how to do it. There will probably always be a need for an assembly level language dialect for each CPU. Hopefully the assembly statements and philosophy would resemble one another, but asking CPU designers to comply to a standard way of doing things each generation is probably asking too much. Look at how long it took to get car bumpers the same height! Yet, as both super computers and microcomputers grow faster and larger, it does seem we could afford to "waste a few cycles" and have machines look alike. By the manufacturer's selling a standardized assembly language interface to a standardized high level language, they also eliminate the need for each installation having a "resident expert" in assembly language hardware interfacing.

CAMAC, a hardware standardization system, is fairly successful in reducing interface problems with hardware. We need a similar system for the software industry to run everything from super computers to microcomputers. The remainder of programming effort would be constrained to a few high level languages that are standardized, having no dialects. Language subsets which do not, in any way, conflict with the standard might be an acceptable deviation for those installations which would find the full language overhead too great a burden. Do we know enough to settle on languages now? No, we probably do not know enough about the programming practice to settle on any standardized languages. To prematurely select a "bad" language could work to slow our learning about good software; yet we still realize on a working day-to-day consciousness that something must be done. For example, new programming students usually learn BASIC or FORTRAN in college. This seems like unnecessary repetition. There is no logical reason why most languages cannot be both interpretive and compiled. There are current needs for both types. One could be given the choice of compiler or interpreter, but leave the language statements alone! If this were done, programs developed interactively with the interpretive system could be compiled and run repeatedly under the other system. With standardized languages, programmers would not have to learn as many languages, or convert programs with each CPU change. The employer could use "trainees and lower grade operatives more extensively".[3] The profession would benefit in much the same way that physicians benefit from using nurses and nurses from using orderlies. The already long training processes would be shortened, freeing

3 IBID

time for productive occupations. These are just some of the many side benefits of standardization of computer languages. Work done on automatic program generation could bring many benefits to the computer industry even if the full automation is never accomplished.

Standardization of operating systems is another area of enormous potential progress towards automatic program generation. It could also have both immediate beneficial and profitable side effects. Everyone recognizes that today we do not have any one operating system good enough for all applications and all sized computers. Much more work will need to be done in studying the similarities of today's successful operating systems. Once these areas are known, the development of new systems can be standardized. We must be very careful not to "freeze" a standard before we know what we're doing, because new and better ideas would have a harder time proving their worth. All I can safely say is that progress must be made and soon. It is wasteful for super computer designers to race feverishly bringing out a product that will not be fully utilized for some years or months. For example, CRAY will begin running this summer, under a practical operating system, at Los Alamos Scientific Laboratory. The first hardware was delivered over two years before. The FORTRAN compiler for the system is still somewhat unreliable.[4] Until operating systems can become portable, that is, it takes less effort to change them, than to rewrite them, new super computers and microcomputers will have to wait for full utilization. "Today's new super computer must

4 Johnson, June 23, 1978

be at least four times faster than the current standard in order to justify the immense effort of conversion, particularly the changes with the new operating system."[5] It would be very advantageous to be able to take smaller steps in progress if only we could afford it. Then, new ideas could be tried one at a time.

Automatic programming may be brought a step closer to reality by using software tools such as pre-compilers. Pre-compilers are programs that allow expansions of a language, converting the input to a language output. Expanding a language by pre-compilers may seem like the same folly as introducing a new dialect or new language, but it is not. Rather, it is a way of hiding dialect idiosyncrasies in a commonly understood language. FORTRAN, for instance, has many dialects, yet still lacks the "IF THEN ELSE" structure. If we were willing to give up using our favorite idiosyncrasies to have commonality and portability by programming in an accepted pre-compiler language, then we would be able to forget our dialect compilers. Software progressed greatly upon the invention and use of such tools. It is now time to develop and use these next layer of tools. We should avoid the self-fulfilling prophecy of the short life span of software. When software is designed to be easier to rewrite than modify, the lifespan shrinks dramatically. When we provide for modification by avoiding dialect advantages and use either a standard language or standard pre-compiler, we will have designed in longevity. Let the pre-compiler take advantage of dialects, not the programmer.

5 Baker, June 23, 1978

Pre-compilers also provide an easy mechanism for testing the usefulness of new programming language structures. Within the last decade, many of these have been invented, but almost no languages have been retrofitted with the new conveniences. If all code were routinely passed through a pre-compiler, several benefits would accrue immediately: first, the old programs could still be recompiled with no changes in the course of regular maintenance; second, the new features could be introduced in the course of maintenance, eliminating the need for rewriting; third, personnel would become accustomed to new ideas gradually - "Trying to implement all of the new structured methodologies at once will generally be a disaster,"[6]; fourth, it will gear the using organization up to prepare for the inevitable changes of converting to one of the new and better languages.

After pre-compilers, the next level of development is to develop automatic specification generation. Some work is being done in this area and several examples will be examined. When a programmer needs to write code, a specification must be laid out. This is usually done mentally, accounting for the lack of concrete research. These processes occur within different minds in different ways, and until something is well understood, it cannot be automated. The clerical functions must be separated from the decision making of what step to take next. Then we can develop a tool to do the clerical work.

6 Yourdon, p 263

> "...specification must be separated from implementation, the separation between these two processes should be a formal operational abstract (i.e., very high level) program rather than a nonoperational requirements specification. Structured programming represents the first results of combining these ideas. It is a special case of a more general two-phase process, called Abstract Programming, in which an informal and imprecise specification is transformed into a formal abstract operational program, which is then transformed into a concrete (i.e., detailed low-level) program by optimization. Abstract Programming thus consists of a specification phase and an implementation (optimization) phase which share a formal abstract operational program as their common interface.[7]

Once it is impossible to both generate precise program specifications from informal specification and generate a program from the resulting specifications, the testing, maintenance, and fine tuning of the program can be done on the specification level, not the computer language level. Large computers of the future will be ideally suited for implementation of this methodology by casting off the restraints of size and speed. The turnaround time will not be from one program change to another, but from one program specification change to another.

---

7 Wile and Blazer, p 705

VI. EXAMPLES OF WORK BEING DONE

Automatic generation of certain accounting programs has already been researched by the University of Pennsylvania (PRYWES). The user is assumed to be only proficient in the field of applications and able to resolve ambiguities, not in computer languages. The user composes statements in a domain specific language. These statements stand alone in containing a "chunk" of information. Statements either describe data or data relations. The language processor requests information or changes as necessary to resolve incompleteness, ambiguities, and inconsistencies. The processor then describes what is the function of the program and generates input for a standard high-level language optimizing the compiler. The user cannot specify the order of evaluation or memory assignments. Thus the user need not be familiar with flow-charting. Statements are independent. Modification of statements can thus be done one at a time. Statements may be entered by a group as the statement information occurs to them. Also, by providing documentation, the processor is able to provide a collection of "programs" and provide the user with a hard copy for reflection and modification.

A prototype automatic program coder for generating business data processing systems is being developed at MIT (RUTH). They are taking the approach of automating steps one at a time with the product of each state being a descriptive representation appropriate for the next stage of development. In this way, each software tool of automation may be used as it is developed. This prototype is designed to handle a restricted yet significant subset of data processing applications. Work has concentrated, not on transforming the natural language specification into

abstract program specification, but on implementing and optimizing a program given the abstract specification. This part has been implemented and is considered operational.

Los Alamos Scientific Laboratory has developed a program code generator based on the MODCOMP computer system called PROGRAM Z (POORE, SUNIER, NELSON). It is not intended to be used by non-programmers, yet provides a great deal of support for those who know some software (FORTRAN) and are familiar with the experiment. In Z, the experimenter can build up a series of FORTRAN calls to a macroprocessor. In real time, each of these calls is read, the appropriate macro found, brought into memory, and executed. Results of the macros are stored in memory allocated by the user. Thus, the user must enter commands in the right order, and have a memory allocation scheme. The system does not correct statements before run time so debugging must occur at run time. Z "programs" may also be executed in batch with up to four users at once. This system saves the programmers a lot of effort since end users do their own "coding".

Whenever machine resources are cheaper than programmer resources, all efforts should be made to eliminate the interface of programmer and user. By eliminating programmers entirely from normal operating modes, this system gives the users only what they ask for, with immediate answers as well. While there is still a pseudo programming language to be learned, it is similar to FORTRAN and directly related to the tasks at hand. We will see a lot more of these "domain specific" languages in the future. Once experience is gained in providing automatic program generation in small

domains of work, we will be in a position to expand the domain. The ultimate domain of all that humans are capable of questioning is of course not necessary even if it were possible.

Los Alamos has also developed a compile-type general-purpose data acquisition system, "Q" (KELLOGG, MINOR, SHLAER, SPENCER, THOMAS, VAN DER BEKEN. The user must be familiar with computers and computer languages in this case, PDP-11 and FORTRAN, respectively. It is useful in a variety of experimental needs for managing data acquisition and display during an experiment. Users describe events and events drive the system. Possible commands include a data taking module, an RSX-11D handler for data recording and distribution, control and computational analysis modules, and histogram entry, retrieval and plotting modules. The need was to allow physicists to design, code, debug, and run their programs - giving them all the tools they would possibly need. Users are allowed to write FORTRAN subroutines for their own purposes and are encouraged to be independent of the staff programmers.

In both the Q and Z systems, Los Alamos had to satisfy the tremendous demand for software generated by putting minicomputers onto an experiment. In the future, this will become increasingly common, not just in experimental laboratories, but in manufacturing, warehouses, and educational facilities. It has been proven many times that once people are shown what computers can do, they will produce a strong demand for using them.

Sperry Research Center has also done work on automatic program generation to discover techniques that will make computers more directly accessible to nonprogrammers (BLACK). Their

> "...Dialogue Processor is a universal facility which has many potential uses; essentially it can front-end any parameter or transaction driven system. It is currently being used as an interactive program specification technique in conjunction with an application customizer. Among other possible applications are its use as an aid to formulating complex command language statements (e.g. JCL) and as a query language interface for data bases."[8]

By having a dialog driven system, learning is reduced or eliminated, user responses can be minimal and validated immediately, question sequence can change dynamically, and the user can save or review previous sessions. By using an "automated consultant" even complex sets of specifications can be generated - especially using forms. Clearly, these types of specification generator, coupled with automatic program coders, are paving the way for automatic program generation.

8 Black, p 397

CONCLUSION

In this paper I have presented some of the methods that will be needed to make science fiction a reality. The hardware of tomorrow will be cheaper, faster, better, and more efficient. Human resources are not going to be able to keep up with the demand generated by the new super computers to super micros if we persist in using these same tired methods. Some of the new methods will be based on increased programmer efficiency. New methodologies are being invented all the time, but new methodologies will not be enough. Programmers and institutions are going to have to learn to share software and software ideas. Public libraries must be set up to provide a forum for new knowledge. New computer languages are going to have to be invented which can request information to resolve ambiguities, which resemble speaking tongues, and which are easier to use. Domain specific automatic program generators will soon be cropping up everywhere to save on valuable programmer resources. Once enough experience is gained in these areas, we will see even complicated programs such as operating systems being generated easily and transported from CPU to CPU easily. Once we have gained the understanding of programming enough to thoroughly automate it, programming can once again afford to become an art.

REFERENCES

1. L. H. Baker, LASL personal conversation, June 23, 1978.
2. R. Balzer, "Whither Automatic Programming," AFIPS National Computer Conference Proceedings (1978).
3. J. Black, "A General Purpose Dialogue Processor," AFIPS National Computer Conference Proceedings (1978).
4. L. M. Branscomb, "The Everest of Software," Computer Software Engineering Polytechnic Press (1976).
5. T. Fay, "Subroutine Libraries," Mini-Micro Systems, May 1978, pp. 65-68.
6. M. Hammer, "The Impact of Automatic Programming Research," AFIPS National Computer Conference (1978).
7. R. T. Johnson, LASL personal conversation, June 23, 1978.
8. M. Kellogg, M. Minor, S. Shlaer, N. Spencer, R. Thomas, H. Van der Beken, "QUAL and the Analyzer Task," Los Alamos Scientific Laboratory, October 1977.
9. S. Krakowiak, M. Lucas, J. Montuelle, J. Mossiere, "A Modular Approach to the Sturctured Design of Operating Systems," Computer Software Engineering, Polytechnic Press (1976).
10. R. C. Linger, & H. D. Mills, "On The Development of Large Reliable Programs," Current Trends in Programming Methodology, Vol. I, Prentice-Hall, Inc. (1977).
11. W. A. Martin and M. Bosyj, "Requirements Derivation in Automatic Programming," Computer Software Engineering, Polytechnic Press (1976).
12. P. J. Plauger and B. W. Kernighan, "Software Tools," Addison-Wesley (1976).
13. R. Poore, J. Sunier, R. Nelson, "Program Z Data Acquisition and Analysis System," LASL, December 1977.
14. N. Prywes , "Automatic Generation of Computer Programs," AFIPS National Computer Conference Proceedings (1977).

REFERENCES CONTINUED

15. G. Ruth, "Protosystem I - an Automatic Programming System Prototype," AFIPS National Computer Conference Proceedings (1978).

16. T. A. Standish, "The Future of Automatic Programming," AFIPS National Computer Conference Proceedings (1978).

17. D. Wile and R. Balyer, "Transformational Implementation," AFIPS National Computer Conference Proceedings (1978).

18. E. Yourdon, "The Choice of New Software Development Methodologies," AFIPS National Computer Conference Proceedings (1977).